Memory for Real-time Applications

Proper handling of memory will improve a real-time application's deterministic behavior. Three areas of memory management within the purview of a real-time application are considered :

1. Memory Locking
2. Stack Memory for RT threads
3. Dynamic memory allocation

Keep in mind that the usual sequence is for an application to begin its execution as a regular (non-RT) application, then create the RT threads with appropriate resources and scheduling parameters.

Memory Locking

Memory locking APIs allow an application to instruct the kernel to associate (some or all of its) virtual memory pages with real page frames and keep it that way. In other words :

- Memory locking APIs will trigger the necessary page-faults, to bring in the pages being locked, to physical memory. Consequently first access to a locked-memory (following an `mlock*()` call) will already have physical memory assigned and will not page fault (in RT-critical path). This removes the need to explicitly pre-fault these memory.
- Further memory locking prevents an application's memory pages, from being paged-out, anytime during its lifetime even in when the overall system is facing memory pressure.

Applications can either use `mlock(…)` or `mlockall(…)` for memory locking. Specifics of these C Library calls can be found here The GNU C Library: Locking pages [http://www.gnu.org/software/libc/manual/html_node/Locking-Pages.html]. Note that these calls requires the application to have sufficient privileges (i.e. CAP_IPC_LOCK capability [http://man7.org/linux/man-pages/man7/capabilities.7.html]) to succeed.

While `mlock(<addr>, <length>)` locks specific pages (described by *address* and *length*), `mlockall(…)` locks an application's entire virtual address space (i.e globals, stack, heap, code) in physical memory. The trade-off between convenience and locking-up excess RAM should drive the choice of one over the other. Locking only those areas which are accessed by RT-threads (using `mlock(…)`) could be cheaper than blindly using `mlockall(…)` which will end-up locking all memory pages of the application (i.e. even those which are used only by non-RT threads).

The snippet below illustrates the usage of `mlockall(…)` :

```
/* Lock all current and future pages from preventing of being paged to swap */
if (mlockall( MCL_CURRENT | MCL_FUTURE )) {
        perror("mlockall failed");
        /* exit(-1) or do error handling */
}
```

Real-time applications should use memory-locking APIs early in their life, prior to performing real-time activities, so as to not incur page-faults in RT critical path. Failing to do so may significantly impact the determinism of the application.

Note that memory locking is required irrespective of whether *swap area* is configured for a system or not. This is because pages for read-only memory areas (like program code) could be dropped from the memory, when the system is facing memory pressure. Such read-only pages (being identical to on-disk copy), would be brought back straight from the disk (and not swap), resulting in page-faults even on setups without a swap-memory.

Stack Memory for RT threads

All threads (RT and non-RT) within an application have their own private stack. It is recommended that an application should understand the *stack size* needs for its RT threads and set them explicitly before spawning them. This can be done via the `pthread_attr_setstacksize(…)` call as shown in the snippet below. If the size is not explicitly set, then the thread gets the default stack size (`pthread_attr_getstacksize()` can be used to find out how much this is, it was 8MB at the time of this writing).

Aforementioned `mlockall(…)` is sufficient to pin the entire thread stack in RAM, so that pagefaults are not incurred while the thread stack is being used. If the application spawns a large number of RT threads, it is advisable to specify a smaller stack size (than the default) in the interest of not exhausting memory.

```
static void create_rt_thread(void)
{
        pthread_t thread;
        pthread_attr_t attr;

        /* init to default values */
        if (pthread_attr_init(&attr))
                error(1);
        /* Set a specific stack size  */
        if (pthread_attr_setstacksize(&attr, PTHREAD_STACK_MIN + MY_STACK_SIZE))
             error(2);
        /* And finally start the actual thread */
        pthread_create(&thread, &attr, rt_func, NULL);
}
```

Details: The entire stack of every thread inside the application is forced to RAM when `mlockall(MCL_CURRENT)` is called. Threads created after a call to `mlockall(MCL_CURRENT | MCL_FUTURE)` will generate page faults immediately (on creation), as the new stack is immediately forced to RAM (due to the MCL_FUTURE flag). So all RT threads need to be created at startup time, before the RT show time. With `mlockall(…)` no explicit additional prefaulting necessary to avoid pagefaults during first (or subsequent) access.

Dynamic memory allocation in RT threads

Real-time threads should avoid doing dynamic memory allocation / freeing while in RT critical path. The suggested recommendation for real-time threads, is to do the allocations, prior-to entering RT critical path. Subsequently RT threads, within their RT-critical path, can use this pre-allocated dynamic memory, provided that it is locked as described here.

Non RT-threads within the applications have no restrictions on dynamic allocation / free.
